ISSN 0265-2919

80p

THE HOME COMPUTER COURSE

19

MASTERING YOUR HOME COMPUTER IN 24 WEEKS

An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Hardware Focus

Acorn Electron We look at the third generation Acorn home computer 370

Software

Make Believe Simulation in the classroom helps keep school costs down and provides valuable assistance to pupil and teacher 366

Basic Programming

Dummy Run We must now create dummy data files for our address book before we can run the program 376

Insights

Call My Bluff Computer games can be much more challenging than Space Invaders or PacMan 361

Best Bet We examine ways in which home computers can be used as an aid to decision-making, both at home and at work 368

Jet Propelled Fast, silent printers that offer full colour are now available for most home computers at a reasonable price 372

Passwords To Computing

Memory Maps We explain how space in memory is allocated to different tasks by the microcomputer's operating system 364

Pioneers In Computing

Double Shuffle A brief history of the tabulator — precursor to the computer 380

Sound And Light

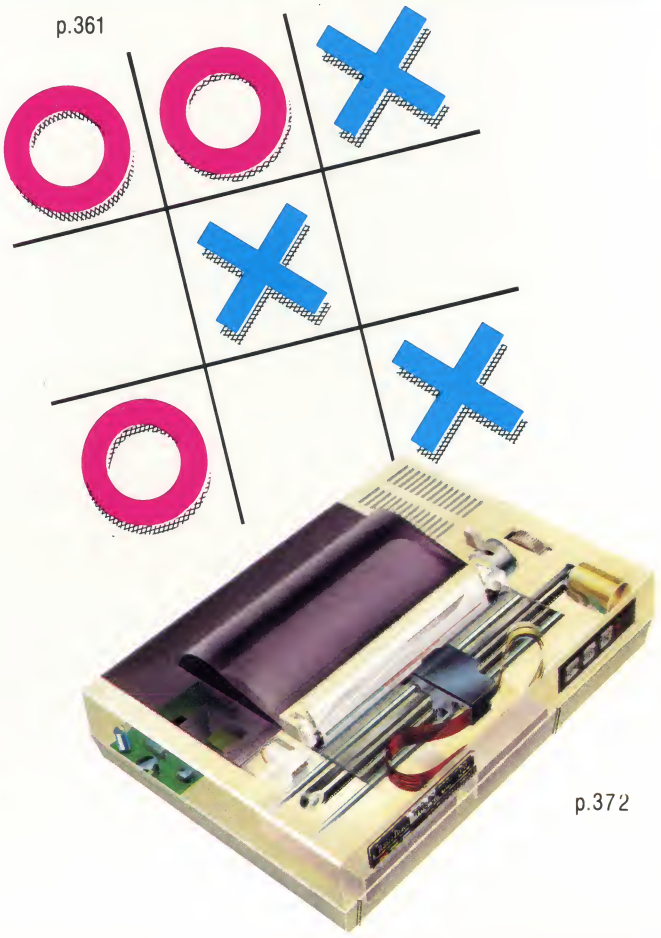
Sound Proof . . . Light Entertainment We look at the Dragon 32's sound generation and further aspects of the BBC Model B's graphics 374

Next Week

- **We examine Memotech's MTX — 16 colours; four sound channels with hi-fi connector; BASIC, LOGO and Assembler as standard, with PASCAL as a ROM-based extra — all for less than £300**
- **Uncommitted Logic Arrays, found in most second generation microcomputers, make the computer designer's job much easier and the end product much less expensive**

A complete index to THE HOME COMPUTER COURSE will appear with Issue 24

p.361



p.372

Editor Richard Pawson; **Consultant Editor** Gareth Jefferson; **Art Director** David Whelan; **Production Editor** Catherine Cardwell; **Staff Writer** Roger Ford; **Picture Editor** Claudia Zeff; **Designer** Hazel Bennington; **Art Assistant** Liz Dixon; **Sub Editors** Robert Pickering, Keith Parish; **Researcher** Melanie Davis; **Contributors** Tim Heath, Henry Budgett, Brian Morris, Lisa Kelly, Steven Colwill, Richard King, Geoffrey Nairns; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Consultant** David Tebbutt; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Chris Cooper; **Production Co-ordinator** Ian Paton; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 85 Charlotte Street, London W1; **© 1983 by Orbis Publishing Ltd: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95
How to obtain your copies of HOME COMPUTER COURSE – Copies are obtainable by placing a regular order at your newsagent.
Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.
How to obtain binders for HOME COMPUTER COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 4, 5 and 6. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595. Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.
Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

COVER PHOTOGRAPH BY IAN McKINNELL: CHESS SET COURTESY OF HARRODS

Call My Bluff

Chess-playing programs are difficult to write, but it is possible even for beginners to construct a simple, 'intelligent' game program

COURTESY OF MILTON BRADLEY LTD

IAN McKINNELL

Invisible Hand
Dedicated chess-playing machines contain the same components as home computers: a CPU, RAM, and the program in ROM, and differ only in the method of input and output. The Phantom, shown here, uses a servo-mechanism and magnets that enable the computer to move the chess pieces automatically. When, for example, a knight jumps over another piece, a sophisticated algorithm is employed that removes any obstruction and then replaces it after the move

Many people, when they begin writing their own computer programs, dream of the day when they will know enough to be able to write a program that plays chess. This is not because chess programs are unavailable, of course. Such programs abound in number, both as packages available for home computers and in the form of dedicated chess-playing machines. But writing chess programs can become an obsession, even among programmers who are not particularly keen on chess as a game. A possible reason for this is that we regard the game as being a highly intellectual pursuit, and therefore a computer that can play chess is a step towards creating an intelligent machine. It would be very difficult to explain to you how to write a complete chess program from scratch, however. But we can explain some of the principles on which computerised 'intelligent' games are constructed, and to a level where you could write a fairly sophisticated program in BASIC.

It should be remembered, however, that the 'games' we are concerned with are not arcade games, adventures or simulations, all of which require different programming techniques and different imaginative skills. We'll begin our discussion of intelligent games with what you might consider to be a trivial example, but one that demonstrates many of the principles of intelligent game writing.

Most children (as well as grown-up children) are familiar with the game Scissors-Paper-Stone. The rules are simple: both players must think of one of these three objects, and then simultaneously hold up a hand in a shape representing the chosen object. The winner is determined according to three rules: scissors beats paper (by cutting), paper beats stone (by covering), and stone beats scissors (by blunting them).

To anyone who has followed the Basic Programming course, it should be a simple exercise to write a program to play the computer's part and keep the score. The RND function is used to select one element from a three-element string array containing 'SCISSORS', 'PAPER' and 'STONE'. The chosen element is then PRINTed when the space bar is pressed. The player types in his own choice (the program relies on his honesty), and the

program calculates who won, displaying the result and an accumulating score for itself and its opponent. If the RND function is truly random, then the scores should even out over a large number of rounds, no matter what strategy the player adopts. Now we need to determine how we can improve the computer's strategy to ensure that it will win over a large number of rounds.

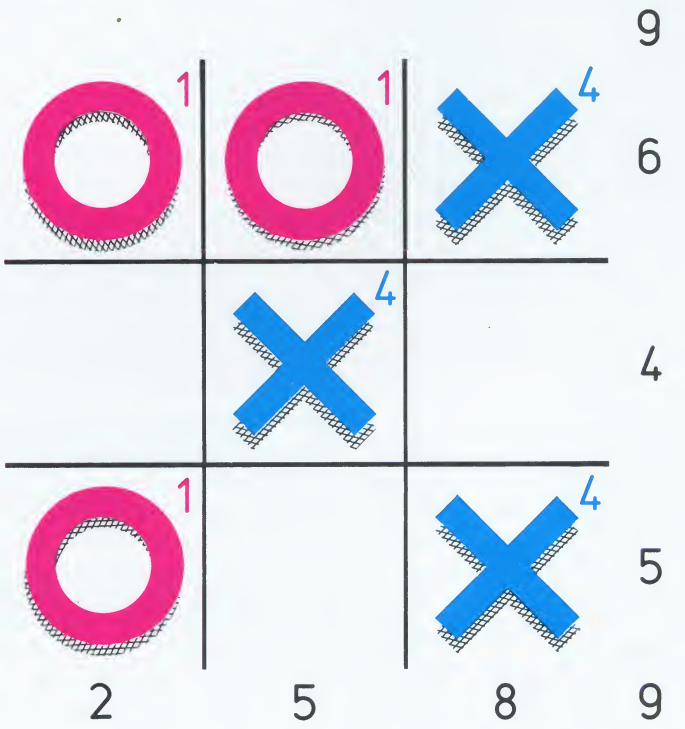
When we looked at random functions (see page 209), we learnt that generating a truly random sequence of numbers is an impossible task for both humans and computers, though the latter make a much better approximation. Over many rounds of our game the human player will invariably favour one of the objects more than the others. You can write a subroutine in your program that keeps track of the player's choices, using an array with three elements called, let's say, CHOICE(1), CHOICE(2), and CHOICE(3). Each time the player makes a choice, one is added to the total in the corresponding array element. The computer can then establish which object is more often presented by its opponent, and play the object that beats this preferred choice.

Three problems arise with this method. Firstly, if the computer consistently plays the same object then the player is very quickly going to take advantage of this. Therefore, the computer must generally be made to choose from the three objects using the RND function, while a routine should be added to ensure that it will more frequently choose the object that will beat the player's most preferred choice.

The second problem is that the player will tend to change his favourite object over the course of the game. So rather than keep a record of his opponent's choices since the start of the game, it would be better that the program simply recorded, let's say, the last 20 choices. This will require a CHOICE array of 20-by-three elements, and a more sophisticated subroutine to add up the three columns and hence predict the best choice for the computer's next turn.

However, the most serious shortcoming of this algorithm becomes apparent if the player deduces the computer's strategy. Then it is relatively easy for him to play in a way that ensures that the computer will lose on more than half the turns. The player could, for example, consistently play the same object, and then switch to another unexpectedly, and so on. What we need is a different algorithm that avoids these problems. Nevertheless, it would be worthwhile developing programs that use both the fully random and the modified random methods, and observing the scores when these are used by unsuspecting players.

Because humans are incapable of making a totally irrational or random decision, it follows that every choice must be a function of the previous choices. That function may be extremely complicated, and the player almost certainly isn't aware of it, but if the computer can work out a good approximation to that function, then it should be able to win fairly consistently. Because each player will have an individual subconscious formula, and will probably change this formula over the course of a long game, the program must be made to interpret the formula while it is playing. Programs that can learn like this are called



Winning Position

'Position evaluation' is fundamental to any board game program — even if the game is as simple as noughts and crosses. In this case, the board is represented as a three by three array, the player's noughts by the value one, and the computer's crosses by a four. Using these values, any position can be evaluated by adding up the totals for every row, column and diagonal. A total of 12 in any of these lines indicates that the computer has won; three means that the player has won; a total of eight shows that two crosses have been played and the computer can win; and so on. The values one and four are used because these ensure that every combination of noughts and crosses gives a unique total

LIZ DIXON

'heuristic' programs.

An heuristic program enables the computer to detect alterations in its opponent's strategy, and modify its algorithm accordingly. Such a program would have to keep a record of, let's say, the last 50 choices of both opponents, in an array. It constantly scans through this track record applying a statistical technique known as 'correlation'.

This involves the computer in making hundreds of comparisons between the player's choice and his previous choice, or the one before that, or the choice made five turns ago. The computer performs the same operation on its own choices. Let's consider the correlation between the player's move and his previous move, for example. We'll call Scissors — element 1, Paper — element 2, and Stone — element 3. First we must set up a three by three array, called say CORR1, because it represents our first correlation test. Now we must work through our game history, looking at the player's choices for the last 50 moves. Every time he followed Scissors (1) by Stone (3), we add one to the element CORR1(1,3); when Stone (3) is followed by Paper (2), one is added to element CORR1(3,2) and so on.

If the player is making truly random choices, then there should be approximately equal values in each element of CORR1 — but this is very unlikely to be the case. So, if the player chose Paper last, then the element in row 2 (Paper) of CORR1 with the largest value will give us the best guess as to what he will choose next. The greater the difference between the elements in any row, the better the correlation is, and the more reliable the prediction will be. However, it is possible that there will be little correlation between the player's choice and his previous choice, in which case we must also perform correlation calculations on the second to last choice, or between the player's choice and the computer's previous choice.

A problem arises if the various correlation routines all predict different results for the player's next move. The program has to decide which is the most reliable advice. In this simple game, all it needs to do is see which test has the most pronounced correlation. For example, the CORR1 array might predict the following probabilities: Scissors 51%, Paper 29%, Stone 20%; whereas CORR2 (which, say, compares the player's choice with the computer's last choice) might give: Scissors 24%, Paper 60%, Stone 16%. Clearly CORR2 has the better correlation, so its prediction should be selected. An intelligent games program will in fact frequently consist of a number of subroutines, each working on different strategies, and each advising the main routine of the best move. The playing routine can regard these subroutines as a 'committee', and act on a majority decision. But as the game proceeds, it can award marks to each routine according to whether its advice was good or not.

If there does turn out to be some correlation between the player's moves or choices and the previous moves of the computer, then it is possible to program in some kind of 'bluffing' factor that will deliberately mislead the player. This works best in gambling games, where the stakes increase as the game continues, and it is worthwhile losing the early rounds to win the later ones.

At the State University of New York at Buffalo (reported in *Scientific American,* July 1978) a collection of poker-playing programs (all of them with a learning capability) were set against each other for several thousand games. The overall winner was a program called the Adaptive Evaluator of Opponents (AEO), which made an initial judgement about the strength of its opponents' hands, and modified this estimate as each game proceeded. The SBI program, 'Sells and Buys Images', did surprisingly badly — its technique was to bluff in order to 'sell' a false image to its opponents, or effectively to 'buy' the playing style of others. The Bayesian Player (BP) tried to make inductive inferences, and improve its play by comparing the predicted consequences of its actions with the actual consequences. Finally, the Adaptive Aspiration Level (AAL) program attempted to mimic a feature believed to exist in human playing: adapting the level of aspiration (that is, the degree of risk it is prepared to take) according to its past record and current status.

No two chess programs or other artificially intelligent routines work in exactly the same way. But by experimenting with the techniques we've outlined here on progressively more complicated games, you may eventually be able to join the exclusive club of chess program writers.

```
  5  CLS
 10  DIM C1(3,3),C2(3,3),C3(3,3)
 20  CR=0
 30  FOR I=1 TO 3
 40  IF C1(PL,I) >CR THEN BG=I:CR=C1(PL,I)
 50  IF C2(PP,I) >CR THEN BG=I:CR=C2(PP,I)
 60  IF C3(P3,I) >CR THEN BG=I:CR=C3(P3,I)
 70  NEXT I
 80  CT=BG-1
 90  IF BG=1 THEN CT=3
100  GET PT: IF PT=0 THEN 100
110  REM LINE 100 WAITS FOR A DIGIT TO
120  REM BE PRESSED.
130  IF CT=PT-1 THEN CS=CS+1
140  IF CT=PT-2 THEN PS=PS+1
150  IF CT=PT+1 THEN PS=PS+1
160  IF CT=PT+2 THEN CS=CS+1
170  CLS
180  PRINT "YOUR CHOICE: ";PT
190  PRINT "MY CHOICE: ";CT
200  PRINT "YOUR SCORE IS ";PS
210  PRINT "MY SCORE IS ";CS
220  C1(PL,PT)=C1(PL,PT)+1
230  C2(PP,PT)=C2(PP,PT)+1
240  C3(P3,PT)=C3(P3,PT)+1
250  P3=PP
260  PP=PL
270  PL=PT
280  GOTO 20
```

Slow Learner

This program, based on the game Scissors — Paper — Stone, illustrates how a program can 'learn' as a game progresses. The computer selects from the numbers 1,2 and 3, compares its choice with the one you have typed in and adjusts the score. The GET statement has been used so that you can simply press the three number keys in rapid succession. If you attempt to make your sequence random, you should find that after a couple of hundred key-presses, the computer's score will pull ahead. It is possible to fool this program and hence continue to win, but more sophisticated routines can be added to it to prevent you from doing this

Map Reading

High-level languages like Basic manage memory automatically; otherwise we need a detailed layout of the memory in order to find our way around the computer

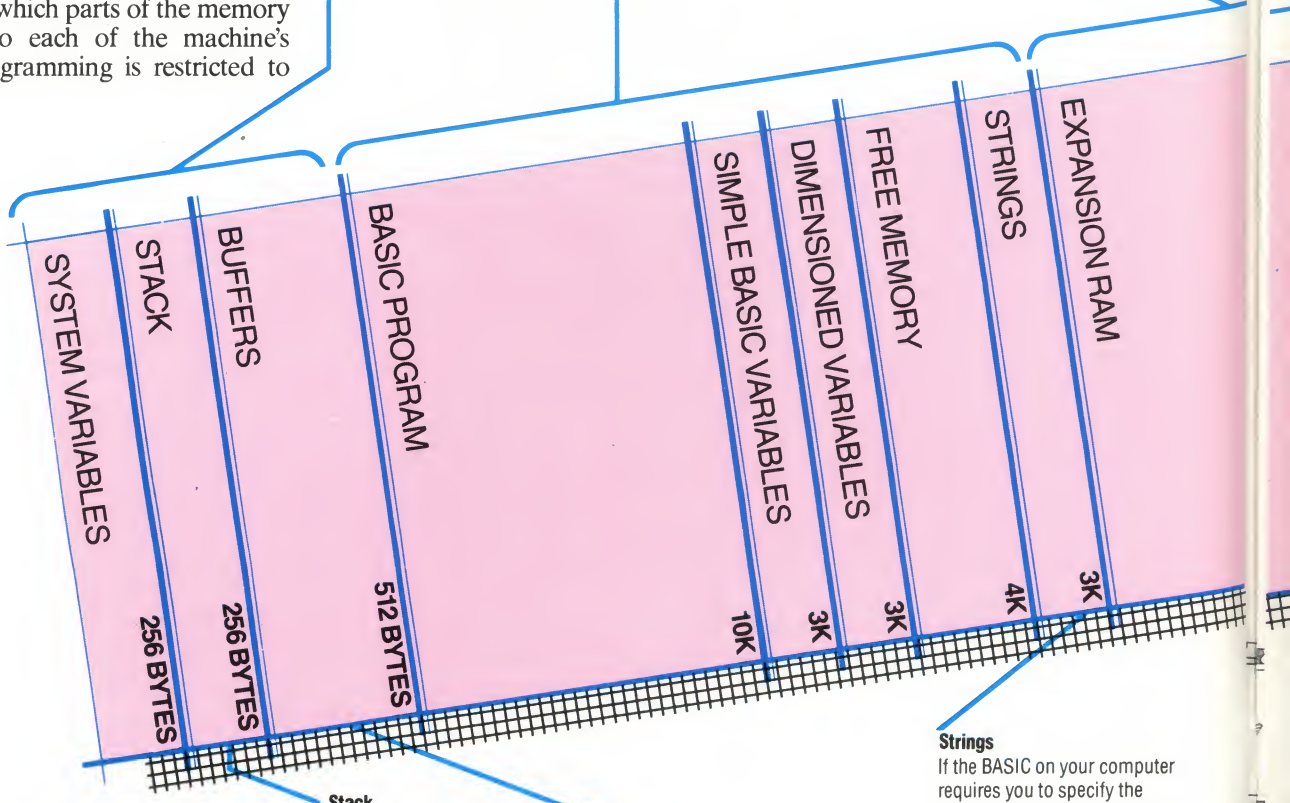
The CPU at the heart of a computer has an addressing range that determines the maximum number of memory locations it can access, and for most home computers this is 64 Kbytes. That memory space must contain all the RAM and ROM that comes with the machine, any expansion RAM or ROM that can be added on, and all the special interfacing chips and ports, which are regarded by the CPU as memory locations as well. One of the most important aspects of the design of a computer is the 'memory map' — the list or diagram that specifies which parts of the memory space are allocated to each of the machine's functions. If your programming is restricted to BASIC then you don't need to know about the memory map in any detail. But if you venture into machine code, or have ideas about building your own hardware add-ons, then it becomes of vital importance.

On these pages we show what a typical memory map contains. Our example is closer to a 6502-based system than one based on a Z80, but most features are common to both. Some manufacturers print a complete map in the user's handbook, while others remain very tight-lipped about the design. However, you will usually find that some user group has managed to work it all out by experimentation.

System Overhead
A computer with 4 Kbytes of RAM may in fact have only 3 Kbytes available to the user for programs. The difference is the system overhead, a section of the RAM that is reserved by the operating system whenever the machine is switched on. Part of this is used for system variables, such as temporary values when computing complex expressions, and pointers to where various things are currently held in memory

User RAM
The size of this determines the sophistication of the programs that you can run, and is perhaps one of the most important considerations when buying a home computer

Empty
Space for expansion RAM must be reserved in the memory map. Some systems permit more than 64 Kbytes to be added on, but this is generally 'bank switched' — a special circuit switches the relevant section of RAM into, and out of, the memory map as needed



Stack
This reserved section of memory is for the exclusive use of the CPU and is organised as a LIFO (Last In/First Out) data structure. A byte can be either 'pushed' onto the top of the stack or 'popped' from the top back into the CPU. When a GOSUB routine is performed in BASIC, for example, the CPU will push onto the top of the stack the location in memory to which it eventually has to RETURN. The stack is extensively used when evaluating arithmetic expressions, and in FOR...NEXT loops

Buffers
A keyboard buffer must be reserved in memory so that characters aren't lost if they are entered faster than the program can process them. A cassette buffer is also required, because most operating systems write data to cassette in blocks

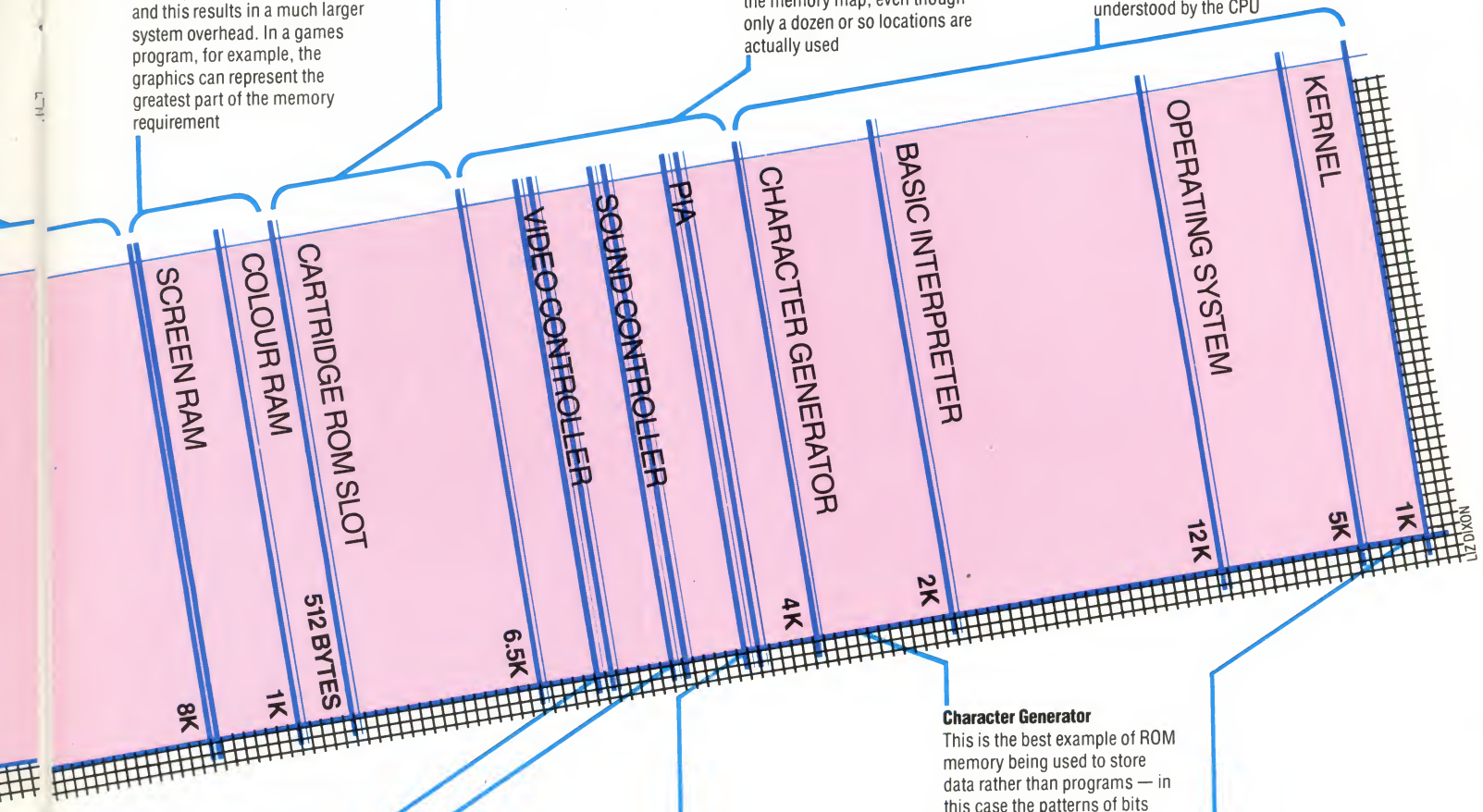
Strings
If the BASIC on your computer requires you to specify the length of all strings in advance, then they will be stored in a table in the same way as dimensioned variables. If, however, it has 'dynamic strings' that can change in length, then the actual data will be stored separately in an area of memory that is constantly changing in size. At intervals, the operating system will instigate a 'garbage collection' that simply cleans up the string area and removes data that is obsolete

System RAM
Some computers have system RAM that is not listed as part of the user RAM. This is generally used for the screen RAM (where one byte corresponds to each character location on the screen) and the colour RAM (where one byte specifies the foreground and background colours for a single character position). Computers with a wide variety of graphics modes and resolutions will need to use memory from the user RAM, and this results in a much larger system overhead. In a games program, for example, the graphics can represent the greatest part of the memory requirement

Empty
When you use a program from a cartridge, it appears in the memory map as expansion ROM. Some machines have spare ROM sockets on the printed circuit board for plugging in additional languages. These will also be reserved in the memory map

Input/Output Chips
The CPU can communicate only with devices that appear as locations in the memory map, so all interface ports and other chips must be included on the map. These will include the interfaces for keyboard, cassette deck, the video controller, and external interfaces such as the printer. The CPU generally addresses memory in the form of blocks (typically 4 Kbytes each). Therefore, the Input/Output chips may occupy 4 Kbytes of the memory map, even though only a dozen or so locations are actually used

System ROM
In a home computer, ROM is used to store information that is always needed and never changes. The most fundamental component of the ROM is the operating system, which is the set of machine code programs that look after the operation of the computer. These programs perform functions such as scanning the keyboard, and storing or retrieving information on cassette. Another component is the BASIC interpreter, which translates programs from BASIC into the low-level instructions understood by the CPU



Video Controller
The most sophisticated graphics, such as sprites and multiple-mode resolution, are increasingly handled in hardware rather than in the software. The video controller(s) will appear in the memory map as a dozen or so single-byte registers, which determine every visual component, from the background screen colour to the exact position of each sprite

Sound Controller
Crude sound effects can be achieved in software, but computers with multiple voices, or with ADSR sound control, invariably have a dedicated sound controller — the output of which is fed into a small amplifier

PIA
Peripheral Interface Adaptors are used to handle most simple interfacing with keyboards, cassettes, joysticks and printers. The most sophisticated chips (such as the 6522 Versatile Interface Adaptor) can convert between parallel and serial data, and have built-in timers, which can be used in programming or to control transmission rates

Character Generator
This is the best example of ROM memory being used to store data rather than programs — in this case the patterns of bits that define how the characters appear on the screen. Some computers allow all or part of the character set to be copied into RAM, and this permits other characters to be defined by the user

Kernel
The 'kernel' (it has a different name on almost every machine) is the heart of the operating system. When the machine is switched on, the CPU will automatically jump to this location and begin executing the kernel program. It will search through the RAM area to determine how much memory is available, and check to see if a program cartridge is plugged in. The kernel also handles the most elementary forms of input and output

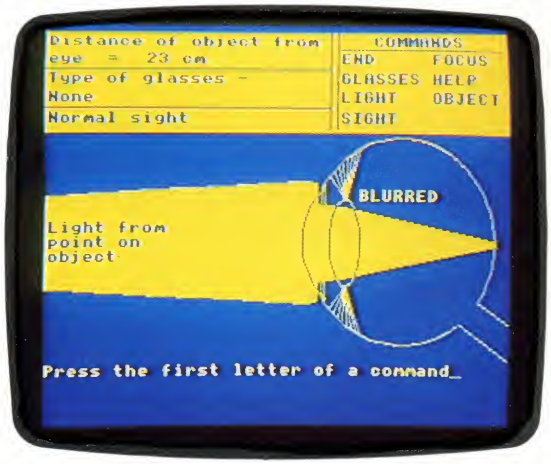
Make Believe

Simulation software allows experiments to be performed without apparatus, specimens or materials, and it is suitable for use at home or in the classroom

Simulation programs, such as the familiar arcade games that put you at the controls of a racing car or aeroplane, are designed to give you an experience as close as possible to the real thing. There is, however, a wide range of simulation software available that aims to educate rather than just exhilarate. Simulation programs are very useful in many areas of school curricula; and especially in those subjects (such as science) where practical experimentation is too dangerous, time consuming, expensive or complicated.

Simulation programs can be used as educational tools in the home as well. For example, in a program called Car Journey, children can use their arithmetic and reasoning skills to 'drive' a car around Britain. Simulation programs are perhaps the most exciting type of educational software currently marketed. Unfortunately, they are available only for a small range of machines — the BBC Micro, the Spectrum, the RML 380Z, and the Apple — the machines most favoured in schools.

Eye



This program demonstrates how the eye works, and how the various parts need to be correctly adjusted in order to see clearly. It simulates the path that light rays take from an object to the retina (the back of the eye where images are formed). You act as the brain, controlling such elements as the object distance, the size of the iris and the focal length of the lens, in order to focus the image on the retina. You are presented with a cutaway diagram of the eye on the screen, with the relevant parts labelled. By using the command LIGHT, the path of a beam of light can be plotted from an object to the eye. If the other variables are correctly chosen you should get the message IN FOCUS. If not it will be BLURRED.

Once you have mastered the functioning of a normal eye, defects such as short sight can be simulated. When it is discovered that it is impossible to focus on a distant object, you must add an extra lens in front of the eye. If correctly chosen, this lens restores normal vision and you have just prescribed your first pair of glasses. Although developed primarily for physics and biology lessons, this program is often used in general computer literacy courses, to introduce computers to young people. Given the simplicity of the subject matter, the excellent error-trapping and the ease of use, it is easy to understand why. The program is produced by Longmans for the BBC Micro.

Ballooning

Ballooning is a home education program for children of eight to twelve years. It is available for the Spectrum from Heinemann Educational Software. The user is at the controls of a hot air balloon, and has to fly it. On the screen you see a cross-section of the countryside, with the balloon initially sitting on the ground. Also shown are four instruments: rate of climb/fall indicator, air temperature gauge, altimeter and fuel gauge. There are just two controls: a gas burner to heat the air and make the balloon rise, and a vent that lets the air escape and the balloon descend.

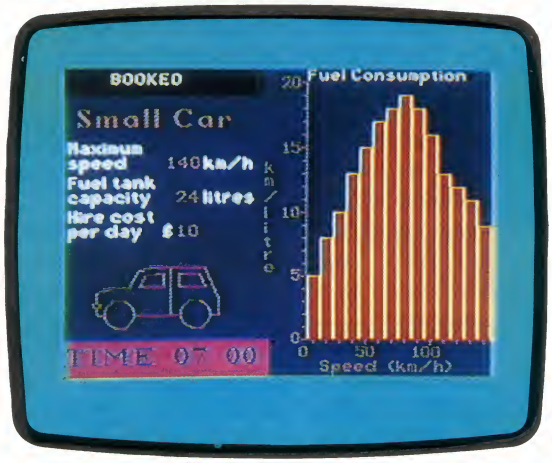
A simple 'flying lesson' teaches the user how to use the instruments and controls to take off, fly and land the balloon. After mastering this, you can fly your own 'mission'. This consists of

landing the balloon at selected locations (marked with an X) where the balloonist receives some instructions. For example, one task is to 'help farmer rescue sheep' — the sheep are to be found in a field marked with an S. If the balloon runs out of fuel, it must land to take on extra gas cylinders.

After a few false starts, crashing into trees and so on, you soon learn how to control the balloon accurately, by using short bursts of the burner. Also, it is not long before you learn to watch the instruments so as to predict when to use the vent or burner. Perhaps the most important benefit is learning to control a system that incorporates a substantial 'time-lag'.

This program, as well as being a realistic simulation, is great fun to use and probably one of the few subjects that has appeal to girls as well as boys.

Car Journey

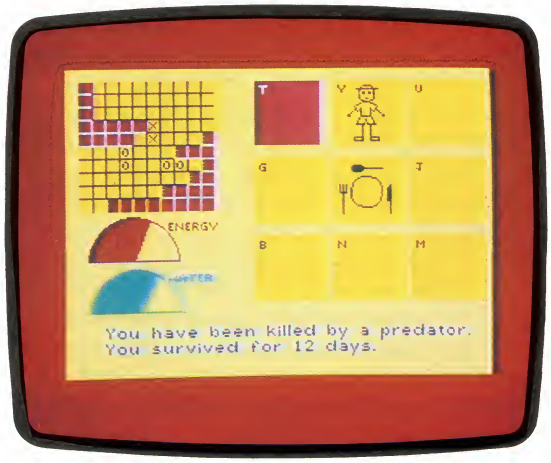


Available from Heinemann Educational Software for the Spectrum, this is a home education program in which the user takes on the role of owner of a small delivery service. Various decisions have to be made with regard to which delivery contract to accept, how fast to drive, and what type of vehicle to use. In doing this the user has to perform calculations involving money, distance, time and even petrol consumption. A map of Britain is displayed, showing 15 cities and the major motorways. A speedometer, milometer, fuel gauge and a clock are also shown.

The first task is to decide which city to start from, and then you have to choose a contract you think you can fulfil from a list of a dozen. For example, one contract is to pick up a consignment of diamonds from Bristol at 1200 hours and deliver it to Dover before 1800 hours on the same day. To do this, you must hire a car, drive it to Bristol, pick up the diamonds and drive down to Dover. If you're successful, you are paid £400, plus a £10 bonus if you are early, and can then choose another contract. Money has to be spent on overnight stops, vehicle repairs, petrol and speeding fines, and if you do not fulfil a contract you incur a hefty £100 fine. If a heavy load is accepted, the car has to be swapped for a larger van, which costs more to hire, consumes more petrol and is slower.

As well as developing a knowledge of vehicles and roads, Car Journey also helps extend the more abstract skills of decision making and logical thought. It even teaches simple economic theory because, in weighing up the pros and cons of a certain contract, the user is performing cost/benefit analysis.

Survival



IAN McKINNELL

If you have ever wondered what it must be like to be a lion (or even a mouse) then Survival is for you. It enables you to play the part of one of six animals (hawk, robin, lion, mouse, fly or butterfly) and experience some of the problems of their day to day existence and the decisions they have to make to stay alive.

The world is represented by a grid of squares on the screen, and you move around this grid (your position being shown as the letter A) by pressing keys on the keyboard. Your main concerns are to find food (the squares marked by an O) and to avoid predators (marked by an X). As you move nearer a marked square, a close-up grid on the right of the screen shows exactly what predator or food you have encountered. Also shown are two meters that indicate how much energy and water you have left. If your energy level gets low you quickly have to find some food, and if the water runs out you have to move next to a blue square (a river); if, however, you accidentally 'fall' into a blue square you will drown.

Some animals have a harder time than others: the butterfly's only source of food is flowers, and these can be difficult to find. The hawk, however, can survive on snails, flies and mice but can fall prey to a human hunter. Through using Survival, you can learn how various species fit into the food chain and appreciate some of the problems faced in surviving in the wild.

Best Bet

Finding optimum solutions to problems is sometimes straightforward, but often it requires advanced mathematics. Computers take the job in their stride

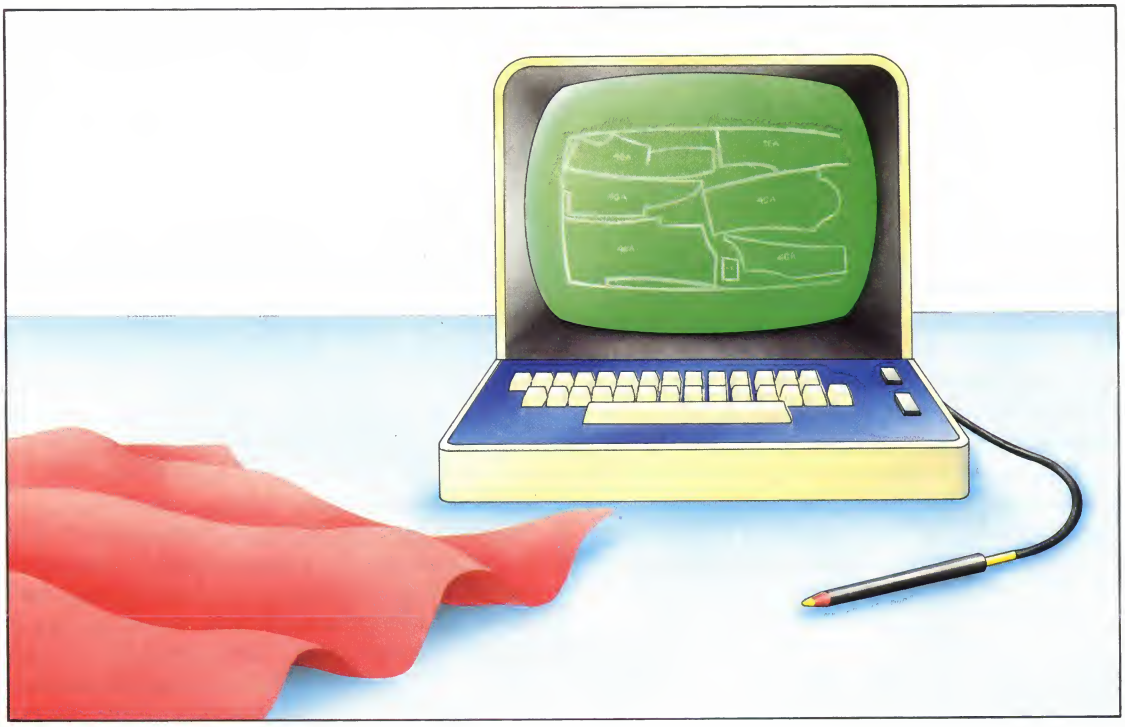
In every decision we make there is invariably a compromise — for example, between cost and effectiveness, or cost and time. We are unlikely to obtain absolute maximum output for absolute minimum cost. The 'optimum' result will fall somewhere between the two.

If we take as an example the choice between two brands of washing powder, the reasoning behind the decision might go something like this: 'If I buy this washing powder, it will cost me 48 pence for 150 grams, but if I buy that one, it will cost 90 pence for 300 grams. But what if I must use 20 per cent more of the less expensive washing powder to obtain the same result? Which brand is cheaper then?' When everything is reduced to a common form — in this case to percentage differences between products — the answer is easy to predict, even before any mathematical calculation is performed.

The concept of 'weighting' a calculation by a constant value is quite normal, and works well when the differences between similar components (the price, for example, or the physical weight), are themselves constant. But when these differences change at different rates, then the mathematics becomes more complex, and we must resort to a form of calculus (in which we solve a number of equations that use the same terms simultaneously) in order to arrive at the right answer. Where the number of terms is small, we might choose to enter them into a matrix, and then manipulate it. Another way is to guess at the answer, and then modify the guess successively until it fulfils all the conditions. Of course, the better the guess, the less time the process will take.

Optimisation techniques such as these are essential to commerce and industry, and are universally applied, especially in manufacturing and construction. Linear Programming, Critical Path Analysis, and PERT (Programme Evaluation Research Technique), are just some of the names given to this optimising method. They predate the computer era by some 30 years in their original forms, and previously required a great deal of manpower to come up with a correct answer in an acceptably short period of time. Applications of this type are quite suitable for home computers, but one should bear in mind that matrix (two-dimensional array) operation requires rather a lot of memory space, and that the matrix arithmetic is in itself quite complex. Fortunately, there are a number of software packages for small microcomputer systems, so the technique is readily available.

One commercial area that has benefited considerably from optimisation is that of clothes



Perfect Fit
Arranging patterns on a sheet of material in order to minimise wastage is a good example of computerised optimisation. One such application is in cutting sheet metal, another is tailoring. Here the computer displays its suggested layout on the screen, and an experienced operator can then make minor adjustments with the aid of a lightpen

ORIGINAL PHOTOGRAPH COURTESY OF BURTONS TAILORING

KEVIN JONES

Diet Optimisation

	PRICE IN PENCE	PROTEIN mg	CARBOHYDRATE mg	FAT mg	CALORIES	QUANTITY REQUIRED
400g POTATOES	10	10	90	0	400	8.5kg
200g SARDINES	50	80	5	5	500	—
400g STEAK	240	150	10	80	1300	—
½ LITRE MILK	28	15	25	20	200	3.75l
MINIMUM REQUIREMENT PER WEEK		250	1000	150	10000	

In this example, the object is to find the optimum combination of four foodstuffs that satisfies a specified minimum dietary requirement at the minimum cost. For this we must tell the computer: the nutritional components (pink) and price per unit (blue) of each foodstuff, and the minimum requirement of each nutritional component for the week (yellow). The computer finds the most critical element and manipulates the rest of the grid around it to find the optimum balance, shown here in green. In this example, the requirements have been satisfied with potatoes and milk only, at the minimum cost of £4.22½ per week. (N.B. this diet is not recommended)

KEVIN JONES

manufacturing. Cloth normally comes in standard units of width — and sometimes of length as well — and the manufacturers' problem is to minimise waste when cutting the cloth, while paying attention to factors like the direction of the nap of the fabric (the way the pile lies).

In one of the most advanced manufacturing tailors in Europe the placing of piece-patterns into a given length of material for the production of made-to-measure suits is worked out using optimising techniques, and the suggested result is shown on a visual display unit. At this point, using object-oriented programming methods (see page 262), the computer operator is requested to exercise his judgement and experience in an attempt to improve on the computer's calculation. The operator makes an improvement, on average, one time in five.

Because the requirements of each job, or each garment, are different, this is an excellent example of the intelligent use of low-level computerised optimisation combined with the experience of the operator. More comprehensive methods are used in industries that repeatedly cut identical objects from sheet material, where the full process of optimisation is allowed to run its course. Because the cutting or stamping operation forms part of a production line, the identical operation will be performed thousands of times. In this case, the cost of the optimisation process divided by the number of units manufactured is more than covered by the savings in wastage.

Critical Path Analysis, as its name suggests, is a method of determining the most important job stream in a manufacturing or construction process — that is, the part of the job with the greatest potential for holding up everything else if it is not completed on schedule. It is very firmly time-based, the period required for the execution of a segment being its value in the CPA diagram or table. Its most common use is during the planning stage of construction projects, so that the builders can allocate men and materials to the various aspects of the project in the right order — plumbing before floorboards, painters after plasterers. Once again, there are software packages available for a wide variety of microcomputers.

While the mathematics of the optimising process may be rather daunting to the untrained, there can be no denying the success and strength of the technique itself. It is one of the few 'number-crunching' tasks commonly carried out on small microcomputers, and is an important component in artificially-intelligent systems, replicating (as it so often does) applied common sense.

Motorway Madness
Apart from social factors, the design and routing of motorways, whether in town or in the country, is very dependent on optimising techniques. The architect will be most concerned with the gradient of hills and sharpness of bends, but the farmer whose land is taken over has a rather different set of criteria. When a new road is being planned a vast amount of data is gathered, which serves to make up a comprehensive model of the situation. This model is then used for a variety of purposes, from graphic representations to route optimisation

COURTESY OF THE MINISTRY OF TRANSPORT

Acorn Electron

In the two years that elapsed between Acorn's BBC Model B and Electron, microcomputer technology has developed dramatically

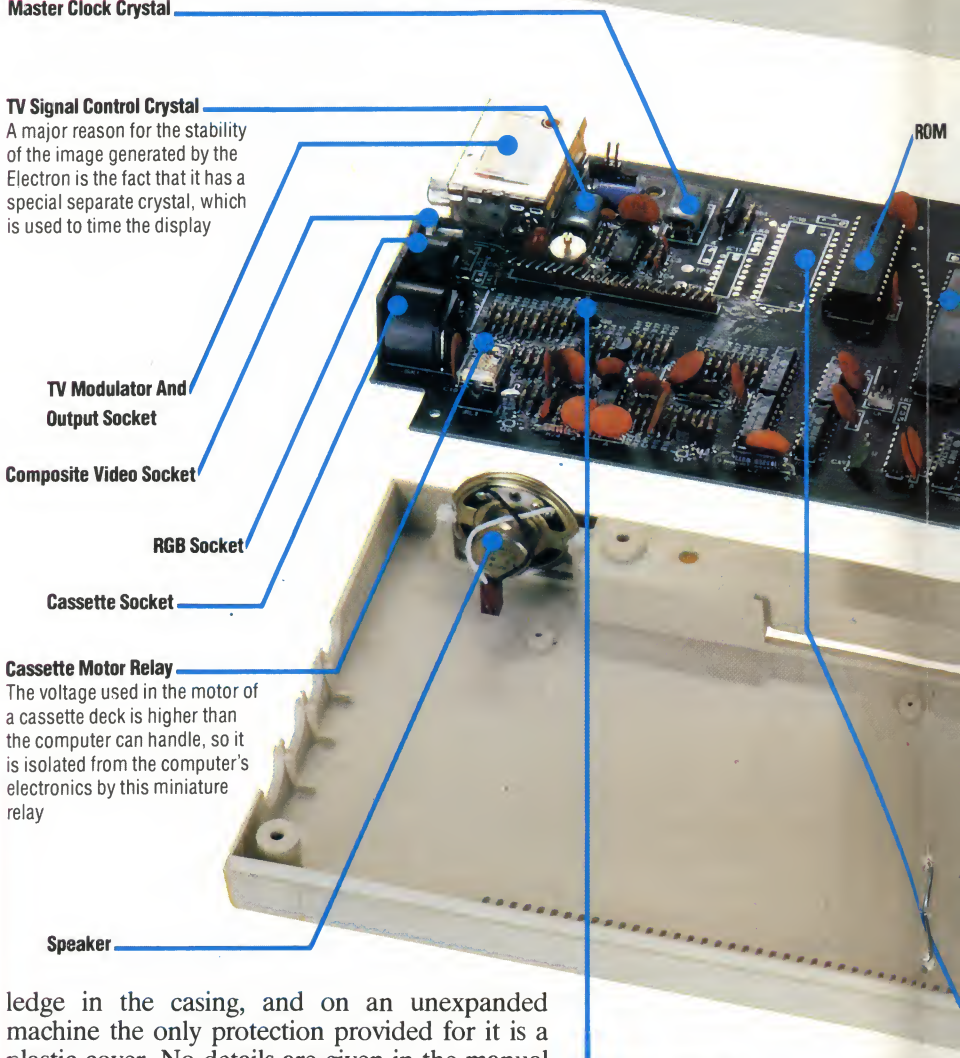
The Acorn Electron is an elegant computer that lives up to its initial impression of being a robust and well designed machine. As a scaled down version of the BBC Micro, it isn't quite as impressive in performance, but feels more comfortable to use. Most of the features of the BBC Micro have been incorporated into the Electron. For example, the SOUND command is used in conjunction with the ENVELOPE command to synthesise different types of musical instruments on both machines.

All of the BBC Micro's graphics modes are available on the Electron, with the exception of Teletext (MODE 7), which is generated in the BBC machine by a special chip. This chip is not available on the Electron's circuit board, and so Teletext-like displays can only be produced by redefining most of the characters and imitating Teletext using MODE 6 (which is, however, restricted to two colours). This is a pity, because the Telextext mode on the BBC Micro is a very economical way of producing quite complex displays without using a lot of memory.

Input and output facilities are also less impressive than on the BBC Micro. Visual output is via TV channel 36, as well as through composite video and RGB sockets to monochrome or colour monitors. But apart from the cassette port there is no immediately usable interface.

Expansion is clearly possible through a large edge connector at the back of the machine. Unfortunately, this protrudes from beneath a ledge in the casing, and on an unexpanded machine the only protection provided for it is a plastic cover. No details are given in the manual about what signals it produces, nor any suggestion as to what may be connected to it. But it is clearly intended that some kind of expansion box will plug into it because there are threaded brass sockets moulded into the casing nearby, which are used to provide a mechanical link between the computer and the add-on.

The built-in BASIC is the now well-known BBC dialect; but this has been considerably expanded and here has many features that make the machine



Master Clock Crystal

TV Signal Control Crystal
A major reason for the stability of the image generated by the Electron is the fact that it has a special separate crystal, which is used to time the display

ROM

TV Modulator And Output Socket

Composite Video Socket

RGB Socket

Cassette Socket

Cassette Motor Relay
The voltage used in the motor of a cassette deck is higher than the computer can handle, so it is isolated from the computer's electronics by this miniature relay

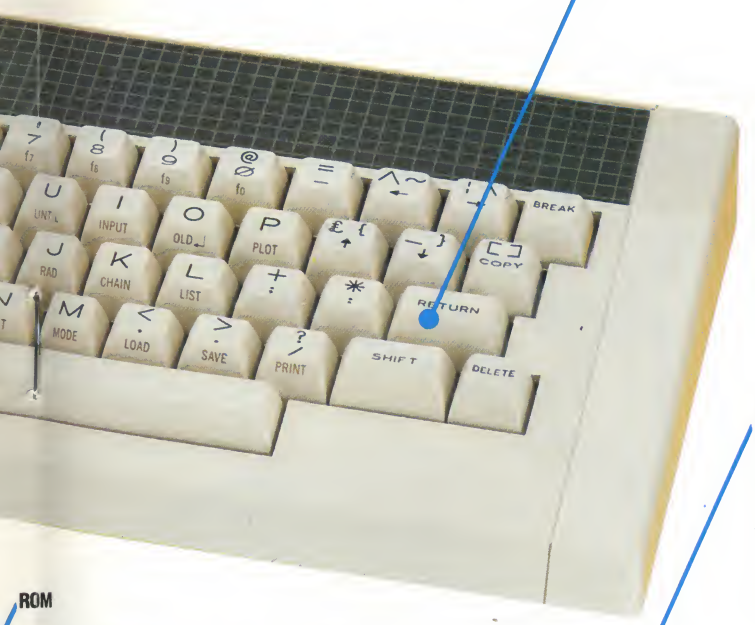
Speaker

Keyboard Connector
The number of pins (22) reveals that the keyboard's output is not decoded into ASCII. If it were decoded, there would be only 10 pins at most (eight for the data, plus the 5v and the ground). This is probably a function of the ULA

JUDY GOLDHILL

JUDY GOLDHILL

Dynamic Duo
The brains behind the Electron were Chris Curry (left) and Herman Hauser (right), who were also largely responsible for the design of the BBC Micro. Curry was a development engineer working for Clive Sinclair, when he employed Hauser. The two men subsequently founded Acorn



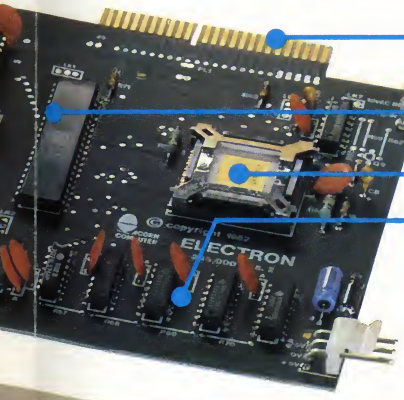
Keyboard
The keyboard is among the best of any home computer, with real typewriter-style keys of very high quality. In practice the keyboard is very similar to that on the BBC Micro. There are no separate function keys, but the same facilities are provided by the Caps Lock key, which if pressed in tandem with a number key, converts it into a function key.

This is extended to the letter keys and three of the punctuation keys, which produce BASIC keywords if they are pressed while the Caps Lock is held down

Expansion Connector
No details of pin values or signal timings are given, but it is obvious that most of the system bus will be available through this connector, as well as TTL and power lines. Therefore, considerable expansion should be possible

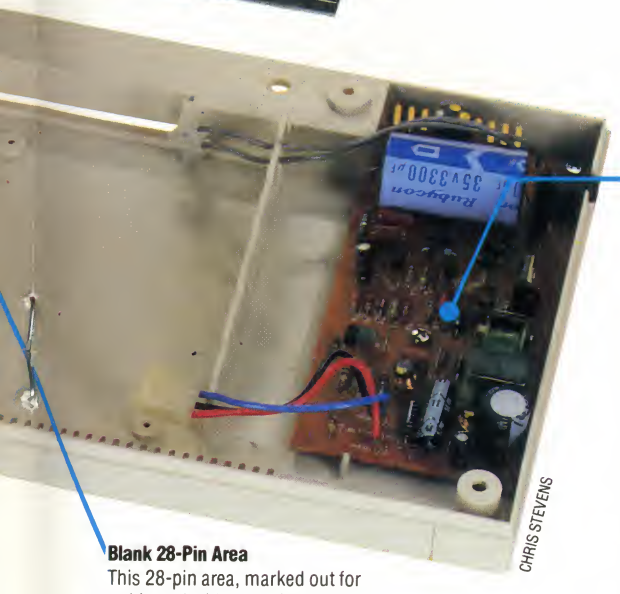
ROM

CPU
Controlling the machine is a standard 6502A processor, clocked at 1.79 MHz. This actually makes the decisions, something which the ULA cannot do by itself



Uncommitted Logic Array
This is the biggest ULA ever manufactured. Apart from the ULA, the 6502 CPU, the ROM and the RAM, there are only nine other chips on the board, all of which are standard TTL logic, each providing just a handful of logic gates

RAM
Bytes are loaded from RAM into the CPU in two halves. First, the lower four bits are accessed (one bit coming from each of the four chips), followed by the upper four. In most machines all eight bits of each byte would be stored in the same chip



Power Conditioning Circuitry
The Acorn Electron is unusual in requiring a 19v AC supply. This has the advantage of being more stable, but needs a more complex circuit to modify it for computer usage

CHRIS STEVENS

Blank 28-Pin Area
This 28-pin area, marked out for a chip and with an unfilled link nearby, would suggest that either additional bank switched ROM may be added, or that a different type of chip may be used

a pleasure to use. Particularly useful is the OSCLI routine, which allows a BASIC program to send commands directly to the operating system, and this permits experienced users to remove some of the constraints of BASIC. The assembler package, which is a feature unique to BBC BASIC, has also been expanded. It has additional keywords for defining variable storage and string printing, both of which are a chore in Assembly language.

In performance the Acorn Electron is better than average. The picture is very steady and sharp, with good clear colour and definition. When some serious expansion facilities, such as disk drives, are available, the Electron will certainly become a justifiably popular machine.

Acorn Electron

PRICE
£199

SIZE
340×160×65mm

CPU
6502

CLOCK SPEED
1.79MHz

MEMORY
64 Kbytes of ROM
32 Kbytes of RAM (with no on-board expansion)

VIDEO DISPLAY
Up to 32 lines of 80 characters. Eight colours with background and foreground independently settable. 127 pre-defined characters and 255 user-definable characters

INTERFACES
Channel 36 TV, composite video, TTL RGB, cassette, system bus (undocumented)

LANGUAGES SUPPLIED
BBC BASIC with in-line assembler

OTHER LANGUAGES AVAILABLE
Should run some other AcornSoft languages such as FORTH and LISP, provided that they are RAM-based. ROM-based languages such as BCPL and PASCAL are incompatible with the unexpanded machine

COMES WITH
Installation and BASIC manual, TV lead, power transformer, introductory cassette

KEYBOARD
56 typewriter-style keys. Single key BASIC keyword entry. 10 user-definable function keys

DOCUMENTATION
Simply excellent. There is plenty of real detail available for the experimenter or the serious programmer. Every BASIC keyword is separately explained; and there is a good section on the Assembly language, which is very important considering the in-line assembler. The functions of the operation system are also well described. Thanks to this wealth of information, most tasks should be relatively easy to accomplish with the machine

Jet Propelled

Full colour printed output is available at a realistic price, thanks to a printer that sprays coloured inks onto the paper, one dot at a time

The different types of printing mechanism available to the home computer user produce print of variable quality. The best results are achieved by full-character impact printers (the daisy wheel is an excellent example of this type); and the poorest reproduction comes from electrostatic and thermal printers. However, the dot matrix printer (see page 74), though noisy and producing typography of only moderate quality, is the most popular system for home computer use.

When printer/plotter devices like the Tandy CGP 115 first appeared, the limitations of the dot matrix printers became more apparent. The printer/plotter machines use miniature ballpoint pens to create complete characters and line graphics on the paper, and these are often in four colours. But the printers most likely to surpass the popularity of the dot matrix printer operate on the principle of firing a stream of microscopic drops of ink in controlled patterns at a sheet of paper. These machines are called 'ink jet' printers.

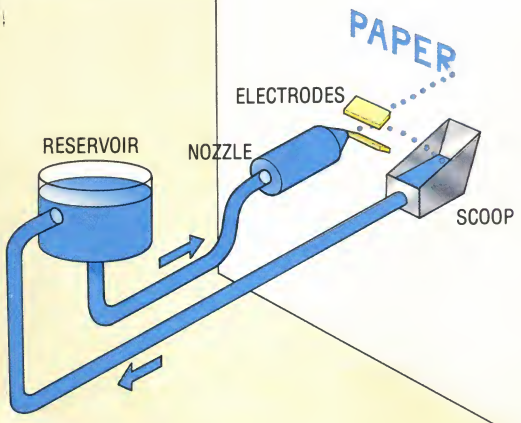
Already well established in the industrial and commercial sectors (alongside the equally sophisticated laser printer), these devices are now beginning to make an appearance on the home computer market. The system works by pumping liquid ink from a reservoir to the tip of a very fine jet. Here minute droplets of ink are charged to a high voltage before being ejected. The valve mechanism is commonly made of piezoelectric material, which allows the droplets to be shaped by very high frequency vibrations.

As the droplet leaves the jet it is suspended by an electric field, which also propels it towards the paper. The sheet of paper is stretched over a sheet of metal (and not a hard rubber roller or platen as it would be with an impact printer). The metal sheet is charged to the opposite potential to that held by the droplet and, as opposite charges attract, helps to pull the ink into the paper. This technique may seem unreliable, but surprisingly little mess occurs. About the worst that can happen is the jet getting clogged or the ink drops becoming oversized.

In principle an ink jet printer works in the same way as a dot matrix printer with only one hammer. The string of ASCII characters arriving at the printer is stored in a buffer until either it is full or a Carriage Return is received. The printer then examines the characters one by one and looks up their corresponding patterns in ROM. Generally, each character will be made up of a number of dots arranged on an eight by eight grid, and the printer builds these patterns up on the paper. It takes eight

Guided Missiles

The first ink jet printers used a more sophisticated system and were very expensive. Inside the nozzle, a piezoelectric device emitted a constant stream of charged ink droplets. These could be guided vertically by two electrodes, as the head moved across the paper. When no mark was required, the droplets could be steered into a scoop and then recycled back into the main reservoir



Priming Pump
This manual pump is used to force ink through the nozzles should they start to become clogged, or simply to get the ink flowing

Circuit Board
This printer contains its own 6809 microprocessor, ROM and RAM. All the incoming data needs to be buffered, because the mechanism prints only one line of dots with each pass of the head

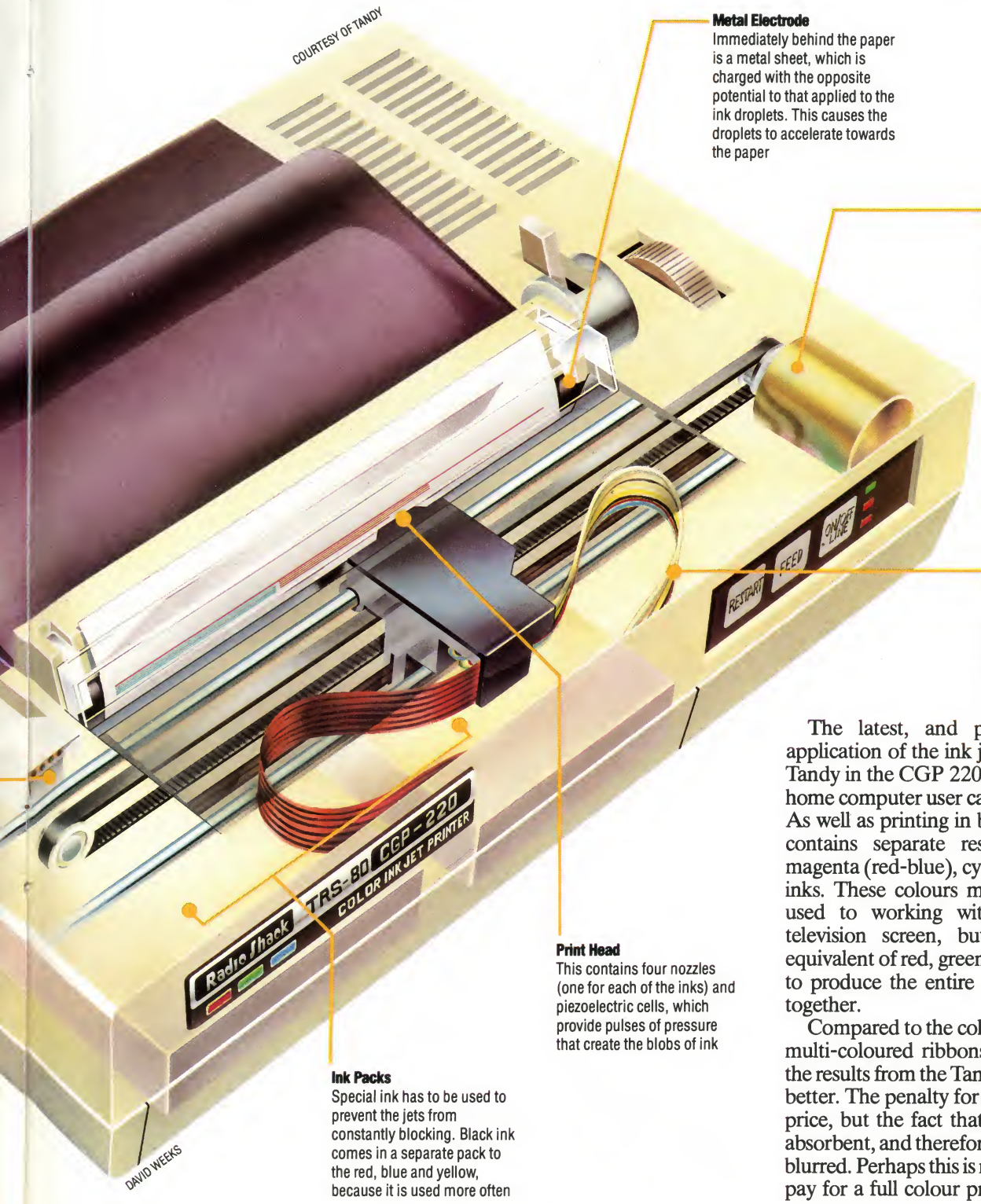
Print Head Lock
An ink jet mechanism is far more delicate than other printing devices, and the head must be locked in the rest position when not in use. The correct operating procedure used immediately after it is turned on is not complicated, but failure to observe it could result in damage to the machine

Sparkling Characters
An interesting variation on the theme of liquid ink jet printers is the 'dry ink' printer. Available both as an Olivetti product and as Acorn's dedicated printer for the BBC Micro, the unit is based on the principle of spark erosion. Printers of this type usually employ a high voltage spark to burn a hole in special silvered paper (the ZX Printer is a typical example). The Olivetti system, however, uses the spark to carry minute particles of carbon from the tip of a replaceable rod to make an impression on the paper.

The printer has several advantages over conventional matrix printers: it is almost silent, the printhead is very light (doing away with the need for powerful motors), and almost any kind of paper will work with the system. The only real drawbacks are that the printing speed is slow, the head prints only one line of dots on each pass across the paper, and the 'ink' tends to smudge

passes of the printing head to create each line of characters, but this is speeded up by allowing the printer to operate in both directions. While the first buffer's worth of characters is being printed out in this way, the next buffer is being filled for printing as soon as the first one is empty. The only difference between the ink jet and the matrix unihammer is that the former fires electrically charged droplets of ink at a page, while the latter imprints a needle through an ink covered ribbon.

In their commercial form, ink jet printers can produce a printed sheet in just a few seconds. The quality of printout, however, can depend on the paper quality: the more absorbent the paper, the more the ink soaks in and blurs the image. At their best, ink jet printers can produce an output quality several times better than that of a dot matrix printer. For large volume business printing they are perfectly adequate. If you need high quality *and* high speed printing, then the laser printer (which works on the same principles as a photocopier) is the only answer.



Metal Electrode
Immediately behind the paper is a metal sheet, which is charged with the opposite potential to that applied to the ink droplets. This causes the droplets to accelerate towards the paper

Drive Motors
The traverse of the print head is achieved with a conventional motor, while the paper advance is driven by a stepper motor — as on a dot matrix printer

Flexible Links
Most printers feature flexible ribbon cables between the PCB and print head. An ink jet has the additional problem of feeding four different inks to a fast-moving device

Print Head
This contains four nozzles (one for each of the inks) and piezoelectric cells, which provide pulses of pressure that create the blobs of ink

Ink Packs
Special ink has to be used to prevent the jets from constantly blocking. Black ink comes in a separate pack to the red, blue and yellow, because it is used more often

The latest, and possibly most interesting, application of the ink jet principle is that used by Tandy in the CGP 220 machine. Here, at last, the home computer user can find true colour printing. As well as printing in black, the Tandy CGP 220 contains separate reservoirs and nozzles for magenta (red-blue), cyan (blue-green) and yellow inks. These colours may be unfamiliar to those used to working with colour graphics on a television screen, but they are the painter's equivalent of red, green and blue, and it is possible to produce the entire spectrum by mixing them together.

Compared to the colour printing achieved from multi-coloured ribbons fitted to matrix printers, the results from the Tandy system are considerably better. The penalty for this is, surprisingly, not the price, but the fact that the paper used has to be absorbent, and therefore the printout can be rather blurred. Perhaps this is not really such a big price to pay for a full colour printout of your work.

Sound Proof

Sound synthesis using the Dragon 32

The Dragon 32 is supplied with only a single square wave oscillator for programming sound, but the wonderfully simple sound commands allowed by Microsoft Extended Colour BASIC enable the construction of music strings that play a passable tune with one command. Unfortunately, there is no means of generating noise. This is very strange as it is difficult to imagine an arcade-type game that does not require noise at some point to make the sound effects interesting.

The SOUND command is useful for sound effects only and the format is as follows:

```
SOUND P,D
```

where: P = Pitch (1-255) and D = Duration (1-255). Pitch is highly inaccurate and bears little relation to a standard musical scale, though middle C can be approximated with the value 89 and reference A at 440Hz is about 159. Duration is similarly inexact but 16 is near to one second, 32 roughly equivalent to two seconds and so on.

This program shows how SOUND can be used for a special effect; in this case, with a little imagination, a UFO taking off:

```
10 FOR P=10 TO 170 STEP 10
20 FOR D=16 TO 1 STEP -1
30 SOUND P,D
40 NEXT D
50 NEXT P
```

PLAY can set an exact pitch, duration and volume for a note. It can also specify a string of such notes to be PLAYed with a selected pause between them at a variable tempo. This makes the construction of tunes with different note lengths and pauses very easy — all PLAYed with this single command:

```
PLAY "T;O;V;L;N;P"
```

where: T = Tempo (T1–T255); O = Octave (O1–O5); V = Volume (V0–V15); L = Length of note (L1–L255); N = Note value (1–12 or note letter); and P = Pause before next note (P1–P255).

It isn't strictly necessary to use the semi-colons between parameters but it would be wise to include them for clarity. The example is very much an arbitrary representation as the parameters can be set in any order. T, O, V, and L retain their values until specified otherwise. In fact, T, O, V, L, and P default to T2, O2, W15, L4 and P0 respectively, unless otherwise specified, so it isn't always necessary to include them in the PLAY statement.

Where timing is involved, as in L and P, the values specified can be thought of as 'notes', and fractions of 'notes' where L1 or P1 is a whole note, L2 or P2 a half note and so on. The actual timing of these is selected by the tempo parameter T, where T1 is slow (a note has a long duration) and T255 is

Light Entertainment

The second instalment of the graphics capabilities of the BBC Model B

BBC BASIC does not provide the full range of high resolution commands that are available on some microcomputers. For example, there are no CIRCLE or PAINT commands. However, it is possible to simulate most facilities using a few lines of BBC BASIC.

The graphics screen has the same co-ordinates regardless of the level of resolution selected, and the axes have their origin in the bottom left-hand corner.The following commands provide control over the graphics screen:

```
MOVEx,y
```

This command moves the graphics cursor to the point with (x,y) co-ordinates, but does not draw a line. Note that the graphics cursor can move completely independently of the text cursor.

```
DRAWx,y
```

As the name suggests, DRAW draws a line from the current graphics cursor position to the point on the screen with the (x,y) co-ordinates.

```
PLOTk,x,y
```

PLOT is a multi-purpose command; its function is governed by the value given to the variable k:

Value of k	Function
0	move relative to last point
1	draw line from origin in foreground colour
2	draw line from origin in inverse colour
3	draw line from origin in background colour
4	same as MOVE
5	same as DRAW
6	same as DRAW but in inverse colour
7	same as DRAW but in background colour

very fast (a note has a short duration). In addition, note lengths can be more flexibly defined by the addition of dots such as L1...or L5. where each dot increases the note length by half its normal value. Therefore L1... = $1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = 2\frac{1}{2}$ notes and L5. = $\frac{1}{5} + \frac{1}{10} = \frac{3}{10}$ note.

There is no absolute way in which the relationship between note and tempo can be represented. The values required can vary for each tune and are best selected by trial and error. This may be a little time consuming but it makes the command very flexible.

The parameter O specifies the octave in which the next note is to be played. O1 starts with C at 131Hz and O5 ends with B at 2093Hz. Middle C begins O2 which is the default octave. Within an octave, notes can be specified in two ways. In the first case a number can be used that corresponds to a musical note as follows:

1	2	3	4	5	6
C	C#	D	D#	E	F
7	8	9	10	11	12
F#	G	G#	A	A#	B

This makes it possible to specify a note as a variable within a selected octave. Alternatively, the required note letter can be used directly to make the statement easier to understand in a listing.

The above explanations are best illustrated with an example. The following command plays F (6) in the default octave O2, for half a note length (L2) at default volume V15. It then pauses for a quarter note length (L1) at volume V20. Tempo is set at T3:

```
PLAY "T3;L2;6;P4;O3;V20;L1;A#"
      < F >  <   A#  >
           pause
```

In addition, the T, O, V, and L parameters can be varied by preset amounts from within the command by the addition of a suffix:

Suffix	Effect
+	Adds one to current value
–	Subtracts one from current value
>	Multiplies current value by two
<	Divides current value by two

The format is: T+, T–, T > or T< for each parameter.

The most useful Dragon facility is the ability to PLAY tunes using substrings. These are first defined, and then PLAYed in any order or repeated:

```
10 A$="F;A#;G"
20 B$="C;D#;F;P4;XA$;"
30 PLAY B$
```

This defines A$ and then includes it in B$ as substring XA$. The resulting tune is C—D#—F—P4—F—A#—G. This technique can be continued as necessary where sequences of notes are repeated a number of times within a piece of music. In all cases the semi-colon following a substring must be included, as in XA$, above.

Higher numbers repeat these eight functions but with extra effects, such as dotted lines instead of solid lines. Values of k between 80 and 87 fulfil a particularly useful function. PLOT80,x,y joins the point (x,y) to the two previously plotted points to form a triangle. The triangle is then filled in with the current foreground colour. This provides the only simple means of PAINTing graphic shapes.

VDU x is equivalent to the more usual BASIC command PRINT CHR$(x). We saw in the introduction to graphics on the BBC Micro that VDU can be followed by a series of numbers. VDU v,w,x,y,z is equivalent to:

```
PRINT CHR$(v);CHR$(w);CHR$(x);CHR$(y);
   CHR$(z)
```

The VDU commands allow the user access to the part of the BBC's operating system that controls graphics and screen display. Although VDU commands may be used within BASIC programs they actually work independently of the language employed. Thus the same VDU commands could be used for a graphics display in PASCAL or any other language offered for the BBC. Each of the BASIC graphics facilities so far discussed can also be implemented by the appropriate VDU command.

Defining characters is very easy on the BBC Micro. VDU 23 controls this function. In the section on user-defined graphics (see page 247) we learned that normal ASCII codes are constructed from a block of eight by eight pixels. The pixels that are visible can be represented by a 1 in binary and those not visible by a 0. Each row of eight bits can then be converted to its decimal equivalent, giving a total of eight decimal numbers to define a character. VDU 23 allows the user to redefine the character with an ASCII code between 224 and 255. For example:

```
10 REM DEFINE A CHARACTER
20 MODE 2
30 VDU 23,240,16,56,124,146,16,16,16,0
40 PRINT CHR$(240)
50 END
```

This short piece of program redefines the character with ASCII code 240 to create an arrow shape. The last eight numbers define this new shape, and line 40 PRINTs the character on the screen.

VDU 24 and VDU 28 respectively control the creation of graphics and text 'windows' on the screen. Using these functions, graphics and text output to the screen can be limited to definable areas. This can be particularly useful when designing interactive programs where a split screen is desirable. All that is required to define a graphics window is to specify the co-ordinates of the bottom left- and top right-hand corners.

MODE 1

This short program listing draws a colourful spiral flower on the screen using MODE 1 resolution. Note the use of filled in triangles to produce the flower petals.

```
10 REM FLOWER
20 CLS
30 MODE 1
40 FOR D=1 TO 3
50 A=600 : B=500
60 MOVEA,B
70 FOR C=1 TO 550 STEP3
80 GCOL0,RND(3)
90 S=(C/(RND(5)+10))
100 X=S*5*SIN(C/16)+A
110 Y=S*5*COS(C/16)+B
120 PLOT85,X,Y
130 NEXT C
140 NEXT D
150 END
```

The spiral pattern is produced by the combination of sine and cosine in lines 100 and 110. Normally this relationship between the x and y co-ordinates produces a circle but the FOR...NEXT loop gradually increases the radius C producing the spiral effect. The co-ordinates of the centre of the spiral, A and B, may be altered to re-position the flower

Dummy Run

In order to use data files it is first necessary to create them in skeleton form, and then fill them with information

At the end of the last instalment of the course, readers were left with the problem of solving this apparent dilemma: how can we make a program read in a file that does not exist (on tape or disk) when the program is first run? The initial activity we are likely to want the program to perform will be to read in the data file and assign this data to arrays or variables. Yet, if we insist on writing to the file first, whenever the program is run, we will have to be very careful in the programming not to lose all the data in the file. As we discovered last time, attempting to open a non-existent file will either simply not work, or else cause the program to 'crash' (stop functioning).

Fortunately, there's a very simple solution. Many commercial software packages include an 'install' or 'set-up' program that has to be run before the program proper can be used, and this is the approach that we shall adopt. Such programs typically allow the user to do a small amount of 'customising' (such as selecting whether the printer to be used will be an Epson or a Brother, parallel or serial, and so on), but they also create data files that will later be used by the main program. Remember, unlike program files, data files can be accessed by any program (see page 316).

To solve our problem and allow *RDINFL* (the routine that reads in the file and assigns the data to the arrays) to be performed, we can write a very simple set-up program that does nothing more than open a file and write a dummy value into it. We will choose a value that can be subsequently recognised by the program proper as not being a valid address book record. A suitable value would be the character string @FIRST, because no name or address, no matter how obscure its origin, is likely to start with this particular string. *RDINFL* will have to be slightly modified so that when it opens and reads in from the file, it tests for this value before going any further. If your computer doesn't have the @ symbol, then you will have to replace it with '!' or another character — as long as this is a string that won't occur naturally in your address book. First, however, here is the set-up program:

```
10 REM THIS PROGRAM CREATES A DATA FILE
20 REM FOR USE BY THE ADDRESS BOOK
   PROGRAM
30 REM IT WRITES A DUMMY RECORD THAT CAN
40 REM BE USED BY *RDINFL*
50 REM
60 REM
70 OPEN "O",#1, "ADBK.DAT"
80 PRINT #1, "@FIRST"
90 CLOSE #1
100 END
```

As mentioned previously in the Basic Programming course, the details of reading and writing files differ considerably from one version of BASIC to another, but the principle is almost always the same. First, the file must be declared OPEN before it can be used for either input or output. Then the direction of data flow is declared, either IN or OUT. Next a 'channel' number is assigned to the file. This allows more than one file to be open and in use at the same time (for the time being, however, we will use only one file). Finally, the name of the file we wish to use must be declared.

Line 70 in the program (left) is in Microsoft BASIC and is similar in principle to the OPEN statements used by most BASICS (BBC BASIC is somewhat different — see page 319). OPEN, of course, declares that a file is to be OPENed and 'O' says that data will be output. #1 is the number we are assigning to the file for this operation; a different file number could be used later if needed. 'ADBK.DAT' is the name we have given to the file.

Line 80 simply writes a single record to the file. The syntax of writing data to a file is usually (in most BASICS) exactly the same as the syntax used for PRINTing, except that the PRINT statement must be followed by the file number — #1 in this case.

Line 90 CLOSEs the file. Files may be left open for as long as needed in the program, but 'open' files are very vulnerable and should be CLOSEd as soon as possible within the program in order to protect the data in them. If, for example, you were to accidentally switch off the computer while the file was open, you could find that data has been lost when you next read the file.

There is some confusion over the way the terms record and file are used in computers, and this confusion is worst when we are talking about databases, on the one hand, and data files on the other. In a database, the file is a whole set of related information. Using the analogy of an office filing cabinet, the file could be a drawer labelled PERSONNEL. This file could comprise one record (a card in a folder) on each person in the company. Each record (card) would contain a number of fields, identical for each record, containing such information as NAME, SEX, AGE, SALARY, YEARS OF SERVICE etc.

If the PERSONNEL file were computerised, all the information would be treated in exactly the

same way conceptually — one file containing many records, each record containing many fields — just like our computerised address book.

A sequential file on a disk or cassette tape, however, doesn't care how the information in it is used or organised by the program. Data files just contain a series of data items, and each individual item of data is called a record. A single record in a data file wouldn't, therefore, normally correspond to a record in the database sense of the word.

It's up to the program to read in records from the data file and assign them to variables or arrays. These variables and arrays need to be organised to form a 'conceptual' record containing a limited set of related information. There is no one-to-one relationship between the records in a data file and the records comprising a database.

Once the set-up program has been run it should never be needed again. In fact, if it ever were run again it would destroy any 'legitimate' data you might have entered in the address book database. We will see why this would happen when we look at the modified *RDINFL* program.

When the program is run it does not 'know' if there is legitimate data in the data file or not. The first thing *RDINFL* does is to OPEN the 'ADBK.DAT' file and read in the first record (or data item). This is not read into an element in an array, as you might expect, but into a special string variable we have called TEST$. Before any other records are read in, TEST$ is checked to see if it contains the string @FIRST. If TEST$ does contain @FIRST, the program knows there is no valid data in the file and so there is no point in trying to read in any more data and assign it to arrays. Consequently, the file can be closed and the rest of the program can continue. Since there is no valid data in the file, the user can do nothing useful until at least one record has been entered and so the value of TEST$ can also be used to force the program to go to the *ADDREC* subroutine so that at least one valid record will be added before anything else can be done.

If, on the other hand, the value of TEST$ is not @FIRST, the program can assume that there is valid data in the file and can start assigning the data to the appropriate arrays. The modified *RDINFL* subroutine follows:

```
1400 REM *RDINFL* SUBROUTINE
1410 OPEN "I",#1,"ADBK.DAT"
1420 INPUT #1,TEST$
1430 IF TEST$ = "@FIRST" THEN GOTO 1530: REM
     CLOSE AND RETURN
1440 LET NAMFLD$(1) = TEST$
1450 INPUT #1,MODFLD$(1),STRFLD$(1),TWNFLD$
     (1),CNTFLD$(1),TELFLD$(1)
1460 INPUT #1,NDXFLD$(1)
1470 LET SIZE = 2
1480 FOR L = 2 TO 50
1490 INPUT #1,NAMFLD$(L),MODFLD$(L),STRFLD$
     (L),TWNFLD$(L),CNTFLD$(L)
1500 INPUT #1,TELFLD$(L),NDXFLD$(L)
1510 REM  SPACE FOR CALL TO 'SIZE'
     SUBROUTINE
1520 NEXT L
1530 CLOSE #1
1540 RETURN
```

Line 1420 assigns a single record from the ADBK.DAT file to the variable TEST$. The next line then checks this to see if its value is @FIRST. If it is, a GOTO is used to jump to the line that closes the file (line 1530) and then the subroutine RETURNs to the calling program. No further attempts are made to read in data. Assuming that there is no valid data in the file, program control will be returned to *INITIL*, which then calls *SETFLG*. All this routine does at the moment is to set the value of SIZE to 1 if TEST$ = @FIRST. The code for *SETFLG* is given below. Note that there are several REMs to allow space for further flag setting should we want to do this later.

```
1600 REM  *SETFLG*
1610 REM  SETS FLAGS AFTER *RDINFL*
1620 REM
1630 REM
1640 IF TEST$ = "@FIRST" THEN LET SIZE = 0
1650 REM
1660 REM
1670 REM
1680 REM
1690 RETURN
```

SETFLG then RETURNs to *INITIL*, which in turn RETURNs to the main program. *MAINPG* then calls *GREETS*, which displays the greeting message. *GREETS* does not need any modification from the previously published version of it.

The next routine called by the main program is *CHOOSE*. A very small modification to the *CHOOSE* subroutine on page 357 will establish a way of forcing the user to add a record if the program is being run for the first time.

```
3500 REM  *CHOOSE* SUBROUTINE
3510 REM
3520 IF TEST$ = "@FIRST" THEN GOSUB 3860
3530 IF TEST$ = "@FIRST" THEN RETURN
3540 REM  'CHMENU'
3550 PRINT CHR$(12)
3560 PRINT "SELECT ONE OF THE FOLLOWING
3570 PRINT
3580 PRINT
3590 PRINT
3600 PRINT "1. FIND RECORD (FROM NAME)"
3610 PRINT "2. FIND NAMES (FROM INCOMPLETE
     NAME)"
3620 PRINT "3. FIND RECORDS (FROM TOWN)"
3630 PRINT "4. FIND RECORD (FROM INITIAL)"
3640 PRINT "5. LIST ALL RECORDS"
3650 PRINT "6. ADD NEW RECORD"
3660 PRINT "7. CHANGE RECORD"
3670 PRINT "8. DELETE RECORD"
3680 PRINT "9. EXIT & SAVE"
3690 PRINT
3700 PRINT
3710 REM  'INCHOI'
3720 REM
3730 LET L = 0
3740 LET I = 0
3750 FOR L = 0 TO 1
3760 PRINT "ENTER CHOICE (1 - 9)"
3770 FOR I = 1 TO 1
3780 LET A$ = INKEY$
3790 IF A$ = "" THEN I = 0
3800 NEXT I
3810 LET CHOI = VAL(A$)
3820 IF CHOI <1 THEN L = 0 ELSE L = 1
3830 IF CHOI >9 THEN L = 0
3840 NEXT L
3850 RETURN
```

Two lines have been added. The first tests TEST$. This variable still contains the value read into it in the *RDINFL* routine. If it is @FIRST we know that there is no valid data in the file and so the only appropriate option is ADDREC, which is number 6. If the test is passed, control is passed to *FIRSTM*, a routine that displays an appropriate message and sets the CHOI variable to 6. When the subroutine returns to line 3530, TEST$ is tested again (it is

bound to pass) and the subroutine RETURNs to the main program skipping the rest of the *CHOOSE* subroutine since it is inappropriate.

You may have wondered why TEST$ is tested twice. This is to prevent the subroutine RETURNing to the wrong point in the program. Without line 3530, the program would continue on down the rest of *CHOOSE*, presenting the choice menu even though it is not needed. It also avoids the use of GOTOs, though IF TEST$ = "@FIRST" THEN GOTO 3850 would work just as well. GOTOs make the program messy and difficult to follow (programs making excessive use of GOTOs are referred to as 'spaghetti coding').

Before going on to look at *FIRSTM*, readers are referred back to *RDINFL* and the GOTO in line 1430. Since we have consistently argued against using GOTO, why has one been used here? It would have been perfectly easy to CLOSE the file and RETURN by simply testing the value of TEST$ in two separate lines. We used a GOTO here instead to illustrate one of the few instances where its use is excusable. This is within a very short and identifiable program segment, and its function is obvious (and made more so by the REM comment). GOTOs should never be used to jump out of a loop (this can leave the value of variables in an unpredictable state), never used to jump out of a subroutine (this will confuse the RETURN instruction unless a matching jump back into the subroutine is used), and never used to jump to remote regions of the program (this makes the program all but impossible to follow).

The *FIRSTM* subroutine is simple and straightforward: the screen is cleared and a message is displayed informing the user that a record will have to be entered. Line 3870 sets CHOI to 6 so that when control is passed back to *EXECUT* the *ADDREC* routine will be executed automatically. The code for *FIRSTM* follows:

```
3860 REM  *FIRSTM* SUBROUTINE (DISPLAY
     MESSAGE)
3870 LET CHOI = 6
3880 PRINT CHR$(12): REM  CLEAR SCREEN
3890 PRINT
3900 PRINT TAB(8);"THERE ARE NO RECORDS IN"
3910 PRINT TAB(8);"THE FILE. YOU WILL HAVE"
3920 PRINT TAB(6);"TO START BY ADDING A
     RECORD"
3930 PRINT
3940 PRINT TAB(5);"(PRESS SPACE-BAR TO
     CONTINUE)"
3950 FOR B = 1 TO 1
3960 IF INKEY$ <> " " THEN B = 0
3970 NEXT B
3980 PRINT CHR$(12): REM CLEAR SCREEN
3990 RETURN
```

The *ADDREC* subroutine, given on page 379, has two small but important changes from the version we encountered before. After the fields have been entered as elements in the various string arrays, the variable SIZE is incremented and TEST$ is set to a null string (see lines 10090 and 10100). SIZE is an important variable used in various parts of the program so that it knows which records are being operated on. SIZE was originally set to 0 as part of the *CREARR* subroutine. Later, in *SETFLG*, it is set to 1 if TEST$ = "@FIRST". This is done so that when *ADDREC* is first executed, the INPUT statements will put the data into the first element of each array. In other words, INPUT "ENTER NAME";NAMFLD$(SIZE) is equivalent to INPUT "ENTER NAME";NAMFLD$(1).

Line 10090 increments SIZE, so that it now becomes 2. If *ADDREC* is executed again, data will be entered into the second element of each array. Finally, *ADDREC* sets TEST$ to " " in line 10100. This is done because a record has now been entered (though not yet stored in the tape or disk data file). If *CHOOSE* is executed again, as it must be to save the data and exit the program, we will not want to be forced to add a new record again. If TEST$ were not cleared, the program would get stuck in an endless loop, and the only way to get out of it would be to reset or unplug the computer, and all the data would be lost.

By setting TEST$ to a null string, the tests in lines 3520 and 3530 of *CHOOSE* will fail and allow the options menu to be displayed. What then happens to SIZE will depend on which routine is executed. So far we have only ensured that SIZE = 1 if there is no valid data in the file, and that this is incremented by 1 each time a record is added. But what would happen if there had been a number of valid records in the file? To answer this we'll have to look at *RDINFL* again.

Line 1420 reads the first data item into TEST$. If it is not @FIRST, it is assumed to be a valid data item. The records in the file are always in the same order, namely: NAMFLD, MODFLD, STRFLD, TWNFLD, CNTFLD, TELFLD, NDXFLD, NAMFLD, MODFLD, etc. If the first record read out is valid data, it must belong in the first element of the NAMFLD$ array, so line 1440 transfers this data from TEST$ to NAMFLD$(1). The next two lines fill up the first elements in the other five arrays. We now know that we have at least one complete (database) record, so SIZE is set to 2. This value must be one greater than the number of valid records read into the arrays, otherwise *ADDREC* would write new data into elements already containing valid data.

Then a loop from 2 to 50 reads the records into all six arrays, incrementing the index L each time round. We have already made the decision to restrict our program to dealing with files of 50 names and addresses, and the DIM statements in the *CREARR* subroutine allocated space for this. However, when you first start using the program, you are unlikely to have a complete file of 50 entries, so we will need a routine in the program that can detect when this is the case, set the variable SIZE accordingly, and abort the reading-in loop.

Consequently, we have included line 1510 to provide a call to a 'SIZE' subroutine, which we will be developing later in the course. There are three ways in which this problem could be handled. First, when we write the data to tape, we could arrange that the first record to be written is the variable SIZE. The *RDINFL* subroutine could then be modified to read in SIZE first and then set up a loop of the form FOR L=1 TO SIZE to read in the records. The second, and preferable, method

(since it doesn't clash with our earlier test for @FIRST in line 1430) is to set up a procedure to be executed after all the records have been written, in which a special flag (of the form @END, perhaps) can be written at the end. A test can then be inserted into *RDINFL* to abort the loop when @END is encountered.

The third method is to make use of the EOF (End Of File) function offered on some computers, which is really an automated version of the second method. These computers have an EOF flag, which is normally set to 0 that is, FALSE but takes on another value (typically 1 to represent TRUE) when the end of file has been reached. Some BASICs allow the EOF flag to be tested as a BASIC variable; in which case, a construct of the form:

```
WHILE NOT EOF(N) (N is the file number)
   DO
      INPUT #N, data to read in)
ENDWHILE
```

will handle the problem. On other machines, the EOF flag is represented as a single bit that must be accessed using the PEEK statement. To find out if your machine has an EOF function, you will need to consult the instruction manual. Because it differs so greatly between machines, we will not be using EOF in our program. But as an exercise, readers might like to attempt to modify the *RDINFL* subroutine for all three possible methods of dealing with files of less than 50 entries.

Generally, it is always a great deal easier to write programs that deal with files of fixed length, but tackling the problem of 'dynamic length' files at this early stage will enable us to modify the program later to cope with files with more than 50 entries.

```
4000 REM  *EXECUT* SUBROUTINE
4010 REM
4019 IF CHOI = 6 THEN GOSUB 10000: REM  SEE
     FOOTNOTE
4020 REM  NORMALLY 'ON CHOI GOSUB etc' --
     SEE FOOTNOTE
4030 REM
4040 REM  1 IS *FNDREC*
4050 REM  2 IS *FNDNMS*
4060 REM  3 IS *FNDTWN*
4070 REM  4 IS *FNDINT*
4080 REM  5 IS *MODREC*
4090 REM  6 IS *ADDREC*
4100 REM  7 IS *MODREC*
4110 REM  8 IS *DELREC*
4120 REM  9 IS *EXPROG*
4130 REM
4140 RETURN
```

The *EXECUT* routine would not normally have line 4019 (hence the odd line number), and line 4020 would normally be either:

```
ON CHOI GOSUB number,number,number etc
```

or a series of:

```
IF CHOI = 1 THEN GOSUB number
IF CHOI = 2 THEN GOSUB number etc
```

Line 4019 is included so that the program will work even though the other *EXECUT* subroutines have not yet been coded .

```
10 REM  'MAINPG'
20 REM  *INITIL*
30 GOSUB 1000
40 REM  *GREETS*
50 GOSUB 3000
60 REM  *CHOOSE*
70 GOSUB 3500
80 REM  *EXECUT*
90 GOSUB 4000
100 END

1000 REM  *INITIL* SUBROUTINE
1010 GOSUB 1100: REM  *CREARR* (CREATE ARRAYS) SUBROUTINE
1020 GOSUB 1400: REM  *RDINFL* (READ IN FILE) SUBROUTINE
1030 GOSUB 1600: REM  *SETFLG* (SET FLAGS) SUBROUTINE
1040 REM
1050 REM
1060 REM
1070 REM
1080 REM
1090 RETURN

1100 REM *CREARR* (CREATE ARRAYS) SUBROUTINE
1110 DIM NAMFLD$(50)
1120 DIM MODFLD$(50)
1130 DIM TWNFLD$(50)
1140 DIM CNTFLD$(50)
1150 DIM TELFLD$(50)
1160 DIM NDXFLD$(50)
1170 REM
1180 REM
1190 REM
1200 REM
1210 LET SIZE = 0
1220 LET RMOD = 0
1230 LET SVED = 0
1240 LET CURR = 0
1250 REM
1260 REM
1270 REM
1280 REM
1290 REM
1300 RETURN

10000 REM *ADDREC* SUBROUTINE
10010 PRINT CHR$(12): REM  CLEAR SCREEN
10020 INPUT "ENTER NAME";NAMFLD$(SIZE)
10030 INPUT "ENTER STREET";STRFLD$(SIZE)
10040 INPUT "ENTER TOWN";TWNFLD$(SIZE)
10050 INPUT "ENTER COUNTY";CNTFLD$(SIZE)
10060 INPUT "ENTER TELEPHONE NUMBER";TELFLD$(SIZE)
10070 LET RMOD = 1: REM  'RECORD MODIFIED' FLAG SET
10080 LET NDXFLD$(SIZE) = STR$(SIZE)
10090 LET SIZE = SIZE + 1 '
10100 LET TEST$ = ""
10110 REM  INSERT CALL TO *MODNAM* HERE
10120 REM
10130 REM
10140 REM
10150 RETURN
```

Basic Flavours

SPECTRUM

Because the Spectrum has the facility for saving or loading whole arrays using the command SAVE-DATA, as explained on page 318, the *RDINFL* subroutine will be completely different — reading in each of the arrays (NAMFLD$, MODFLD$ etc.) in succession. When we begin writing the data in the next instalment, we will publish a complete version of the relevant subroutines for this machine. In the meantime, as an exercise, Spectrum owners can tackle the problem of how to create the dummy file containing @FIRST, as well as determining how many valid entries there are in the array, when reading the file in.

Sinclair machines do not accept program line numbers above 9999. In the full Spectrum listing that will appear in Issue 23 the ADDREC subroutine begins at line 4200 and line numbers increase in steps of 10

See 'Basic Flavours' page 319.

OPEN
CLOSE

TONY LODGE

Double Shuffle

Herman Hollerith and James Powers both developed tabulating machines. Their rivalry dominated the world of computing for six decades

James Powers
Powers' machines were purely mechanical and dedicated to a single application. He nevertheless provided fierce competition for Hollerith

COURTESY OF I.B.M.

Herman Hollerith
Hollerith invented the electromechanical card reader, which was later developed into the tabulator

The machines that Herman Hollerith (see page 240) invented to process the results of the 1890 United States census developed into a range of general purpose data processing equipment known as 'tabulators'. Until the introduction of the first commercial computers in the 1950's, tabulators were essential to the growth of industry and business. In Pittsburgh in the 1930's, for example, a leading department store experimented with a system of customer accounts in which 250 terminals throughout the store were connected by telephone lines to a central bank of tabulators. Goods were priced with punched tags and the information was automatically sent to the tabulators, which then recorded the sale and prepared an invoice for the customer. When the customer's credit rating had been checked, authorisation for the sale was sent to the terminal through an 'on-line' typewriter.

Business competition, in fact, provided the initial stimulus for the development of tabulators. Hollerith's monopoly over the provision of census equipment was broken in 1910 when the Census Bureau invited James Powers to provide alternative machines. Powers offered a system of tabulators that were totally mechanical and therefore did not infringe the patents of Hollerith's electromechanical devices. The rivalry between the two men, and the two companies they later formed, spurred on the growth of data processing machines.

In 1902, Hollerith designed a plug board (rather like a telephone cord switchboard), which could select the columns of the punched card that were to be added up and then output. In this way, Hollerith's machine had a programming capability that his rival's machines lacked; Powers always produced machines dedicated to specific applications. In 1924, Powers patented a way of representing alphanumeric data on punched cards by using a single hole in each column for a number, and a combination of holes to represent a letter. Hollerith quickly responded with his own system: the now standard 80-column card. Each column of this card contained 12 rows of holes that were 'read' by wire brushes completing an electrical circuit with a metal contact beneath the card. Some advanced systems used a light detector for this purpose.

The first tabulators could only count or accumulate totals, but later more advanced mathematical functions were provided for manipulating data. Unlike computers, which were invented by scientists for mathematical purposes, the tabulator was created to be an information processor. People were quickly inspired to work out applications for the new machines. Special tabulators were adapted for use in computing tables, in wave analysis, and in astronomy — tabulators identified the planet Pluto in 1930. Tabulators eventually became sophisticated enough to deal interactively with large amounts of data — IBM patented one that could keep records on the transactions of 10,000 bank accounts. But their greatest impact was in collating data on a scale never seen before.

Tabulator Machines

The tabulator in its heyday in the early 1950's consisted of eight separate units. Data was put onto each card by a 'card punch', which could process 200 cards an hour. A separate 'verifier' checked the accuracy of the punch operator, and when the cards became worn a 'reproducing punch' created new copies. An 'interpreter' printed an explanation of the data above each column for easy reference.

The 'tabulator' itself accumulated totals of data in the columns, and output the results at a rate of 9,000 cards per hour. This tabulator was often connected to a 'multiplying punch' that provided more sophisticated mathematical functions. The 'collator' could compare the data in two stacks of cards or merge two stacks together. Finally, the 'sorter' could take a stack of cards and sort them into 13 piles — one for each of the 12 holes, and one for a blank column.

The operation of the tabulator could be changed with control codes (in the 11th and 12th positions), and control cards were brightly coloured to mark them out in a stack. When a control card was encountered, the tabulator would begin a new operation — such as counting a different field. In census work, an example of a field would be the data relating to a house, or a street, or a city. At each change of field the tabulator would print out a subtotal — in our example this would provide the population of each house, street or city. Some of the techniques of data processing were carried over from tabulators into the early computer languages

BBC HULTON PICTURE LIBRARY

Mentathlete

Home computers. Do they send your brain to sleep – or keep your mind on its toes?

At Sinclair, we're in no doubt. To us, a home computer is a mental gym, as important an aid to mental fitness as a set of weights to a body-builder.

Provided, of course, it offers a whole battery of genuine mental challenges.

The Spectrum does just that.

Its education programs turn boring chores into absorbing contests – not learning to spell 'acquiescent', but rescuing a princess from a sorcerer in colour, sound, and movement!

The arcade games would test an all-night arcade freak – they're very fast, very complex, very stimulating.

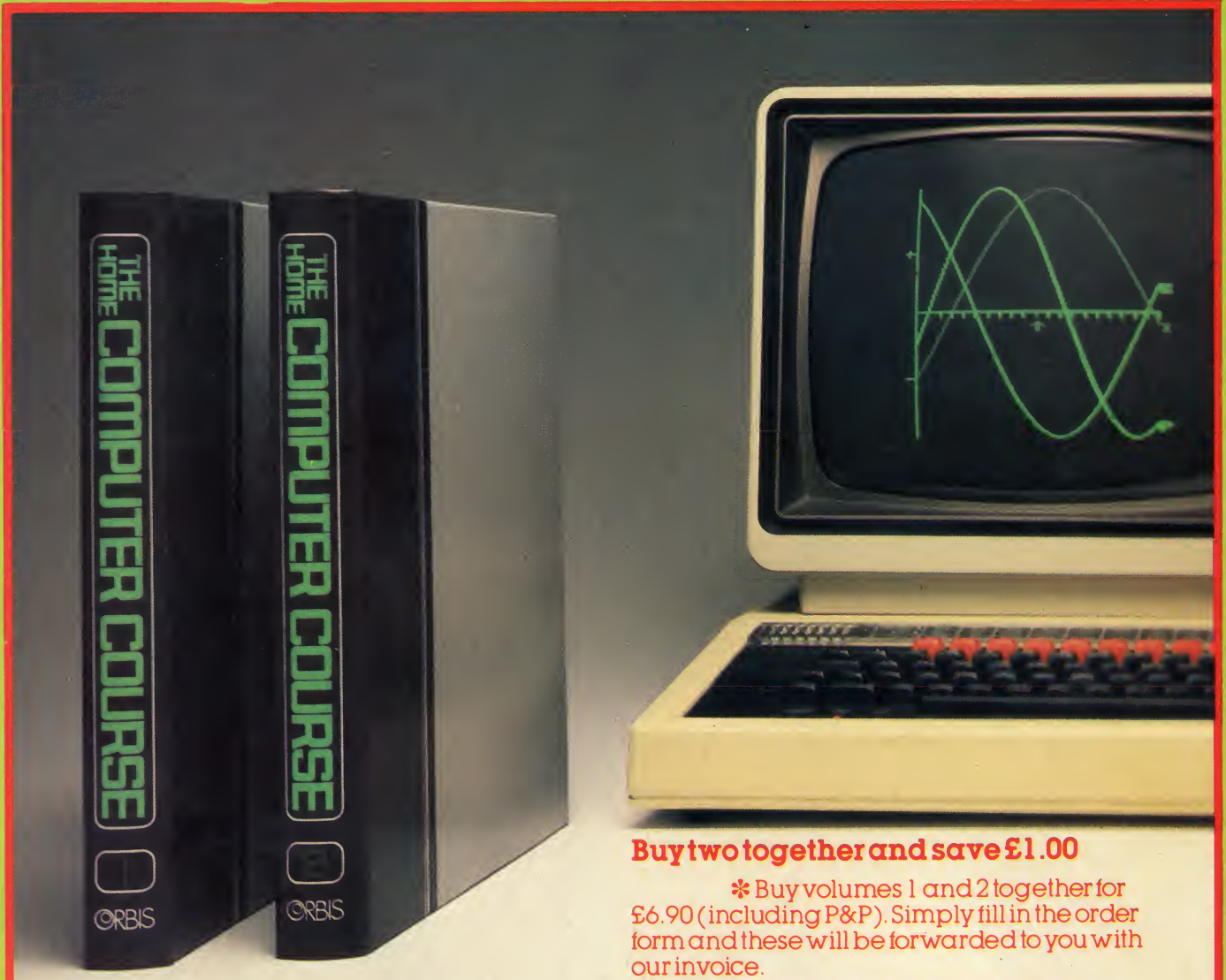
And the mind-stretchers are truly fiendish. Adventure games that very few people in the world have cracked. Chess to grand master standards. Flight simulation with a cockpit full of instruments operating independently. Genuine 3D computer design.

No other home computer in the world can match the Spectrum challenge – because no other computer has so much software of such outstanding quality to run.

For the Mentathletes of today and tomorrow, the Sinclair Spectrum is gym, apparatus and training schedule, in one neat package. And you can buy one for under £100.

sinclair

THE HOME COMPUTER COURSE BINDER



Now that your collection of Home Computer Course is growing, it makes sound sense to take advantage of this opportunity to order the two specially designed Home Computer Course binders.

The binders have been commissioned to store all the issues in this 24 part series.

At the end of the course the two volume binder set will prove invaluable in converting your copies of this unique series into a permanent work of reference.

Buy two together and save £1.00

✻ Buy volumes 1 and 2 together for £6.90 (including P&P). Simply fill in the order form and these will be forwarded to you with our invoice.

✻ If you prefer to buy the binders separately please send us your cheque/postal order for £3.95 (including P&P). We will send you volume 1 only. Then you may order volume 2 in the same way – when it suits you!

Overseas readers: This binder offer applies to readers in the UK, Eire and Australia only. Readers in Australia should complete the special loose insert in Issue 1 and see additional binder information on the inside front cover. Readers in New Zealand and South Africa and some other countries can obtain their binders **now**. For details please see inside the front cover.

Binders may be subject to import duty and/or local tax.

THE LAST WORD IN LOGIC